Interpolation search in edgelists of graphs

Damjan Angelovski¹, Kristijan Petrovski¹, and Ljupco Kocarev¹,²

¹ Macedonian Academy of Sciences and Arts, Skopje, Macedonia.
² Faculty of Computer Science and Engineering, UKIM, Skopje, Macedonia

damjangelovski@gmail.com {petrovski.kristijan, lkocarev}@manu.edu.mk

Abstract. Most applications with graphs use adjacency lists to represent them in memory. The purpose of this work is to put edge lists as a viable alternative. In order to achieve that, the algorithm of interpolation search is used. Assuming that the values in the array are uniformly distributed (i.e. there is no prior knowledge about the array), it is proven that the number of iterations has a mean and variance bounded to \( \log_2 \log_2 E \) (where \( E \) is number of directed edges). Its performance is measured and compared for edge lists of graphs with different properties. Generally, higher average degree gives better results and Erdos-Renyi graphs outperform power law graphs. Additionally, the algorithm is evaluated with timing random walks on real and generated graphs.

Keywords: Interpolation search · Graph · High performance computing · Big data · Edge list.

1 Introduction

Graphs in memory can be represented in several ways: Adjacency list, Edge list, Adjacency matrix and Incidence matrix. When working with large, sparse graphs the adjacency and incidence matrix are not very practical as they occupy a lot of memory. For those types of graphs adjacency or edge lists have to be used.

While working with graphs queries like “Is a node \( A \) connected to a node \( B \)?”, “How many neighbors does a node \( A \) have?”, “Who are the neighbors of \( A \)?” are typically asked. To find the neighbors of \( A \) in an adjacency list, one needs to just go to the index of \( A \) in an array to get the singly-linked list of neighbors, while an edgelist needs to be searched. This is why most applications use the adjacency list approach. If the edgelist is searched for \( A \) using linear or binary search we will need \( O(E) \) and \( O(\log_2 E) \) iterations respectively, where \( E \) is the number of edges in the graph. Instead, when interpolation search is used, the number of iterations is lowered to less than \( \log_2 \log_2 E \), which hopefully makes the sorted edgelist more appealing in real-world applications and comparable to cutting-edge research on graph computation like X-Stream [1] and M-Flash [2].

Interpolation Search is a method for finding a certain value in a specified, sorted array. Assuming that the values in the array are uniformly distributed (i.e. there is no prior knowledge about the array), it is proven that the number of iterations has a mean and variance bounded to \( \log_2 \log_2 E \) [3,4]. The performance
of interpolation search is calculated when used in edge lists of graphs with certain parameters (average degree and degree distribution). Experimental results show improvement in the mean and variance and their correlation with the graph parameters.

2 Algorithm

The interpolation search algorithm is based on the regula falsi (false position) method from numerical mathematics. It combines the bisection method (binary search) and the secant method. It is an improvement on the secant method because it always converges and an improvement on the bisection method because it converges faster, using the values as a heuristic.

![Fig. 1: An example for one iteration of the regula falsi method, where the starting interval is $[x_1, x_2]$ and the search value is zero. The cut value $m$ is $x_3$ and the new interval is $x_3, x_2$ because $0 > f(x_3)$.](image)

According to it, for a given interval $[a, b]$ (where in programming it is usually the first and last index of the given array) and a search value $v$, a linear function is constructed which passes through the points $(a, f(a))$ and $(b, f(b))$ (where in programming $f(x)$ corresponds to the value of the array at index $x$). Next the value $m$ is calculated, for which the linear function has a value of $v$. $v$ is compared with $f(m)$ and if it is different, a new interval is set. $[a, m]$ in case $v < f(m)$ and $[m, b]$ in case $v > f(m)$. With iterative repetition of this procedure the interval is shrinking, so the required value will eventually be found in the array, if it exists. An example of one iteration can be seen at fig. 1.

In order to use the method on a sorted array, $m$ needs to be approximated to an integer value. In this work the floor function is used to achieve this. Additionally an improvement is made by using $\text{ceil}(m)$ when $m$ is not an integer and $\text{floor}(m)$ is equal to the lower bound of the interval.

In real-world applications the algorithm needs to output the index in the array where the searched value is found, which is of no use to this research. So
the algorithm outputs the number of iterations, which is the objective of this research.

<table>
<thead>
<tr>
<th>Interpolation search algorithm for searching through a given array <code>edgelist</code> for a given value <code>searchValue</code>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initializing the variables, <code>lowIndex</code> and <code>highIndex</code> on the first and last indexes of <code>edgelist</code>, <code>lowValue</code> and <code>highValue</code> on the respective values of <code>edgelist</code>. <code>numberOfIterations</code> on zero.</td>
</tr>
<tr>
<td>2. Repeat while <code>highIndex &gt; lowIndex + 1</code></td>
</tr>
<tr>
<td>- <code>midIndex</code> = <code>(highIndex - lowIndex) * (searchValue - lowValue) / (highValue - lowValue)</code></td>
</tr>
<tr>
<td>- if <code>midIndex</code> isn’t an integer and <code>floor(midIndex) == lowIndex</code></td>
</tr>
<tr>
<td>• then <code>midIndex += 1</code></td>
</tr>
<tr>
<td>- <code>midIndex = floor(midIndex)</code></td>
</tr>
<tr>
<td>- if <code>searchValue == edgelist[midIndex]</code> then break from the loop</td>
</tr>
<tr>
<td>- if <code>searchValue &gt; edgelist[midIndex]</code></td>
</tr>
<tr>
<td>• then <code>lowIndex = midIndex</code> and <code>lowValue = edgelist[midIndex]</code></td>
</tr>
<tr>
<td>• else <code>highIndex = midIndex</code> and <code>highValue = edgelist[midIndex]</code></td>
</tr>
<tr>
<td>3. return <code>numberOfIterations</code></td>
</tr>
</tbody>
</table>

3 Method

It can be noted that when searching an edgelist for a beginning vertex, the ending vertex is irrelevant. For that reason the ending vertex is not generated in the tests, so the edgelist is represented only with a one-dimensional array. For testing the algorithm an array of length (number of edges \( E \)) \( 2^{30} = 1073741824 \) is used. The array is sorted in ascending order, in order to be able to be searched with interpolation search.

The graph is taken to be directed because an undirected graph has to have each edge twice in the sorted edgelist, for an optimal search, thus practically making it directed. For generating the edgelist 12 values for the average degree of the graph were taken (1.5, 2, 3, 4, 6, 10, 15, 20, 30, 50, 70, 100). Those values are actually the number of edges (the length of the edgelist) divided by the number of vertexes (the range of values in the edgelist). With that the number of vertexes \( N \) is dependent on the number of edges \( E \) and the average degree.

Generating the edgelist is done with four different methods:

1. The degree distribution is modeled in such a way that the probability of a vertex to have a certain degree is following a geometric distribution. This is the characteristic of power law graphs. The parameter of the geometric distribution is calculated in such a way that the number of edges and vertexes have to be satisfied and there are no vertexes without neighbors. With that the probability that a vertex has a degree \( k \) is \( (1 - p)^{k-1} \cdot p \), where \( p \) is equal to the reciprocal value of the average degree. The number of vertexes for
a given degree is an integer, approximate to the corresponding probability multiplied by \(N\).

The edgelist is generated in such a way that the index of the vertex and the corresponding degree are independent. So the indexes in order are given a degree randomly, according to the remaining degree distribution. The degree of a vertex with a given index is actually the number of times that index appears in the edgelist.

2. The second method is similar to the first, with the difference that instead of a geometric distribution, a Poisson distribution is taken and vertexes without edges are taken into account. With that the probability that a vertex has a degree \(k\) is \(\frac{\lambda^k e^{-\lambda}}{k!}\), where \(\lambda\) is equal to the average degree. Well known graphs that have this property are Erdos-Renyi graphs.

3. The edgelist is generated directly. Every element in the array is set to a random integer from a discrete uniform distribution in the interval \([0,N-1]\) and then the array is sorted.

4. The fourth method is a continuation of the third. Additionally the vertexes with no neighbors (whose indexes do not appear in the edgelist) are removed i.e. all indexes appear at least once in the edgelist, thus effectively reducing the range of values in it.

For every combination of average degree and generating method, 100 tests were made. For every test an edgelist is generated and an interpolation search is made for the value of every element in it. Mean and variance are calculated cumulatively for all 100 tests. Additionally the edgelist is divided in \(2^{20} = 1048576\) intervals with \(2^{10} = 1024\) elements and a mean for each is kept.

## 4 Results

<table>
<thead>
<tr>
<th>avg. deg.</th>
<th>test 1</th>
<th>test 2</th>
<th>test 3</th>
<th>test 4</th>
<th>avg. removed vert.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>4.2040</td>
<td>4.5842</td>
<td>4.5897</td>
<td>4.1821</td>
<td>1.5711 1.6 E+08</td>
</tr>
<tr>
<td>2</td>
<td>4.4877</td>
<td>4.5052</td>
<td>1.5893</td>
<td>4.9444</td>
<td>4.5855 1.0406 7.3 E+07</td>
</tr>
<tr>
<td>3</td>
<td>4.5678</td>
<td>4.1720</td>
<td>1.0817</td>
<td>4.1682</td>
<td>4.0154 0.9340 1.8 E+07</td>
</tr>
<tr>
<td>4</td>
<td>4.5999</td>
<td>1.7532</td>
<td>3.9815</td>
<td>0.9073</td>
<td>3.9785 0.9070 4.9 E+06</td>
</tr>
<tr>
<td>6</td>
<td>4.6269</td>
<td>1.8269</td>
<td>3.7502</td>
<td>0.7376</td>
<td>3.7406 0.7362 3.7286 4.3574</td>
</tr>
<tr>
<td>10</td>
<td>4.6174</td>
<td>1.8890</td>
<td>3.4817</td>
<td>0.5836</td>
<td>3.4760 0.5906 3.4760 0.5946 4872.33</td>
</tr>
<tr>
<td>15</td>
<td>4.5954</td>
<td>1.9226</td>
<td>3.2796</td>
<td>0.5086</td>
<td>3.2894 0.5049 3.2894 0.5050 21.9</td>
</tr>
<tr>
<td>20</td>
<td>4.5717</td>
<td>1.9398</td>
<td>3.1633</td>
<td>0.4591</td>
<td>3.1693 0.4569 3.1693 0.4569 0.12</td>
</tr>
<tr>
<td>30</td>
<td>4.5547</td>
<td>1.9559</td>
<td>3.0137</td>
<td>0.4098</td>
<td>3.0160 0.4086 3.0160 0.4086 0</td>
</tr>
<tr>
<td>50</td>
<td>4.5194</td>
<td>1.9690</td>
<td>2.8360</td>
<td>0.3782</td>
<td>2.8384 0.3766 2.8384 0.3766 0</td>
</tr>
<tr>
<td>70</td>
<td>4.4953</td>
<td>1.9747</td>
<td>2.7330</td>
<td>0.3645</td>
<td>2.7225 0.3653 2.7225 0.3653 0</td>
</tr>
<tr>
<td>100</td>
<td>4.4607</td>
<td>1.9788</td>
<td>2.5949</td>
<td>0.3517</td>
<td>2.5966 0.3509 2.5966 0.3509 0</td>
</tr>
</tbody>
</table>

Table 1: The cumulative means and variances from all tests.
From Table 1 and fig. 2 it can be observed that the means and variances of all tests are below $\log_2 \log_2 E = 4.90689$. In power law graphs with the increase of the average degree, the mean rises to 4.629, until the average degree of 6 and then drops to 4.4607 whereas the variance rises from 1.0727 to 1.9788. In Poisson degree distribution graphs with the increase of the average degree, the mean falls from 4.5842 to 2.5949 whereas the variance rises to 1.5893 for an average degree of 2 and then falls to 0.3517.

The second and third test give the same results because it is practically the same test. It is well known that using a discrete uniform distribution multiple
times results in binomial distribution for the probability that a certain number appears a given number of times. Also the binomial distribution converges towards the poisson distribution and can be replaced under conditions which are satisfied here.

The results of the fourth test are an improvement over the previous two. This is more evident in the graphs with lower average degrees, because there more of the vertexes have a degree of zero and are removed (see last column of table 1)

Fig. 3: The probability density of the means of the $2^{20}$ intervals from the geometric and Poisson degree distribution tests

On fig. 3 we can see the distributions of the results of the first two tests. The number of iterations from a single interpolation search is an integer, but on the figures we have the probability density of the $2^{20}$ means of intervals with $2^{10}$ elements. In the power law tests it is noticeable that with increasing the average degree, the centers widen and their y-coordinate increases up to average degree of 6 and then decreases. This matches fig. 2. In the poisson degree distribution tests with increasing the average degree, the y-coordinate of the centers decreases, but their shape behaves more chaotically than what is expected from fig. 2. This can be due to the fact that these results are based on means of the results and not the results themselves.

On fig. 4 the interval means are shown according to where they appear in the edgelist, for 3 different average degrees (1.5, 6 and 100) for the first two tests. The visualizations generally match the means and variances from fig. 1. It can be noted that searching for values that are near the beginning or end of the edgelist is faster. This also explains the larger bottom tails in fig. 3.
5 Evaluation

This method is evaluated by running a random walk with $10^6$ nodes 1000 times and calculating the average and variance of the execution time. This is done on different graphs. The first test was on random generated uniform (Erdos-Renyi) graphs with $2^{30}$ edges and different average degrees, same as in the previous method.

The second test was done on real graphs from Wikipedia [5] (12 150 976 nodes and 378 142 420 edges), Friendster [6] (65608366 nodes and 1 806 067 135 edges).
edges) and Twitter [7] (41 652 230 nodes and 1 468 364 884 edges). Those are all directed graphs, so a big random walk is very likely to get stuck in a dead end, so in that case we select a random node for continuing, while not counting it. Then the indexes of the nodes of the graph were randomly relabeled, in order to remove any bias between index and degree of a node, and the tests were run again.

The programming language is C++, compiled with g++ (GCC) 7.3.0 (cygwin) with no additional compilation flags. None of the tests were parallelized and they ran on a single CPU. The tests were executed on a PC with the following specifications:

- Windows 10 pro
- intel i7-4790 @ 3.6GHz, 3601Mhz, 4 core(s)
- RAM: 16GB 1600MHz DDR3

The results from the first test can be seen on table 2 and fig. 5, while the results from the second, on table 3 and fig. 6. It isn’t evident on fig. 6, but the Friendster results lean towards their lower bound. The process of the first test used up 8 192.7 MB of RAM, while the second used up 2 885.9 MB for Wikipedia, 13 779.4 MB for Friendster and 11 203.2 MB for Twitter.

6 Discussion

The means and variances from all tests are far below the so far proven bound of \( \log_2 \log_2 E \) [3] [4]. Generally they decrease with the increase of the average degree, with the exceptions of: the power law graphs variance, the power law graphs means up to avg. deg. of 6 and the variance of the Poisson degree distribution graphs up to avg. deg. 2.
It can be assessed that the reason for the improvement of the results from the \( \log_2 \log_2 E \) bound is the fact that the values in the list are integers, whose range is smaller than the length of the edgelist by a factor of the average degree (sometimes bigger in the fourth test). Also the Poisson degree distribution graphs give better results than the power law ones because there most of the degrees are grouped around the average degree, in contrast to the power law graphs, which have a lot of small degree vertexes and a few big degree vertexes. This makes the power law edgelists deviate more from a linear function, which worsens interpolation search performance.

The results from the random walk on the generated Erdos-Renyi graphs (fig. 5) were unexpected, as the time increased with the average degree, while the number of iterations decreases (fig. 2). This is probably due to the increase of the average degree, which increases the time for acquiring the first and last edge from the node, after the initial search that returns one edge. The results from the random walk on real graphs (fig. 6) were also partly unexpected, as the average times aren’t proportional to the number of edges (E). This is probably due to the structure of the graphs and subsequently, their edgelists. The tests with relabeled nodes gave better results in all graphs, which confirms that the original graphs have a bias between the index and the degree of the nodes.

7 Conclusion

When it comes to real world applications this method has a slightly larger computational complexity than the standard adjacency list approach, but it uses less memory. If \( N < 2^{32} \) it requires only 8 bytes per edge to function and doesn’t need any sort of indexing or hashing. This allows for fitting very large graphs in working memory, which is considerably faster than having to work with HDDs and SSDs.

This approach can be further developed by implementing the standard algorithms for graphs to work with it, with a focus on minimizing memory, and comparing them to cutting-edge research on graph computation.

The results from this work can not only be applied to edgelists of graphs, but in other fields where arrays of similar properties are used.

References


6. The largest connected component of the Friendster social network, as of June 2011 [https://snap.stanford.edu/data/com-Friendster.html] Last accessed 24 Jul 2018